Program Slicing

Nicholas Cameron

Program Slicing

- Overview and example
- Motivation
- Types of slicing
- Implementation
- Tools
- Tool demo - Bandera
- Summary and further reading

Program Slicing

- Debugging technique
- A slice consists of all statements that affect the values at a point of interest
- Produces reduced, executable program
  - value at point of interest unchanged
- More difficult for certain features:
  - control flow (procedures, goto)
  - pointers/arrays
  - object oriented programs
  - concurrent programs

Example

```
1:    f(int x)
2:    {
3:          int y := 25;
4:          String z := "";
5:          for (int i:=0; i<x; ++i)
6:          {
7:                z := z ++ " " ++ y;
8:                y := y + 2 * i;
9:          }
10:
11:         print(x ++ ": " ++ z ++ " " ++
  y);
12:  }
```


Example: (11, {y})

```
1:    f(int x)
2:    {
3:          int y := 25;
4:          String z := "";
5:          for (int i:=0; i<x; ++i)
6:          {
7:                z := z ++ " " ++ y;
8:                y := y + 2 * i;
9:          }
10:
11:         print(x ++ ": " ++ z ++ " " ++
  y);
12:  }
```


Example (11, {y})

```
1:    f(int x)
2:    {
3:          int y := 25;
5:          for (int i:=0; i<x; ++i)
6:          {
8:                y := y + 2 * i;
9:          }
11:         print(x ++ ": " ++ z ++ " " ++
  y);
12:  }
```

Example (cont.)

(11, {x})

```
f(int x)
{
  print(x ++ ": " ++ z
      ++ " " ++ y);
}
```

(11, {z})

```
f(int x)
{
  int y := 25;
  String z := "";
  for{int i:=0; i<x; ++i)
  {
      z := z ++ " " ++ y;
      y := y + 2 * i;
  }

  print(x ++ ": " ++ z
      ++ " " ++ y);
}
```


Motivation

- Debugging is hard: finding the bugs is hard
  - Too much 'noise'
- Weiser noticed programmers automatically filter out irrelevant statements whilst trying to find a fault
- Automation of this process: program slicing

Applications

- Debugging
- Comprehension
  - Maintenance and evolution
- Cohesion measurement
  - And other metrics
- Other uses suggested
  - Compiler tuning
  - Testing
  - …

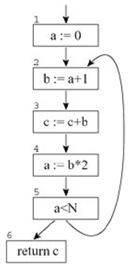
Types of slicing

- Forward vs Backward
- Chopping
  - Slice consists of statements that 'transmit an effect' from source to target
- Static vs Dynamic
  - Static slice: no assumptions regarding input
  - Dynamic slice: for a given input
- Syntax preserving vs amorphous
- Others: quasi-static, conditional, dicing, barrier slicing, etc.

Dataflow analysis

- Weiser's implementation uses dataflow analysis
- General technique widely used by optimising compilers
- Works on a control flow graph: an intermediate representation of a program
- Analyse program flow and variable assignments
- A semantic analysis

Control Flow Graph



Weiser's Slicing Algorithm

- Iterative algorithm
- Notation
  - Slicing criterion: C = (n, V)
  - $i \to_{cfg} j$ means there is an edge from i to j in the control flow graph
  - Def(i) is the set of variables defined in a statement i
  - Ref(i) is the set of variables referenced in a statement i
- Example: `4:  a := b + 1`
  - Def(4) = {a}
  - Ref(4) = {b}

Weiser's Slicing Algorithm

- Find R^0, the set of *directly relevant variables* for each node in the control flow graph, i
- Work back through graph finding relevant variables
- *Directly relevant statements*, S^0 found from R^0
- A branching statement b is *indirectly relevant* if $i \in S^0$ and i is in the range of influence of b, Infl(b)

Weiser's Slicing Algorithm

- We continue by calculating the *indirectly* relevant variables, R^k
  - R^{k-1} and variables affecting $b \in B^{k-1}$
- And *indirectly* relevant statements, S^k
  - B^{k-1} and statements defining R^k
- The fixpoint of S^k is the desired program slice

Weiser's Slicing Algorithm

- $R^0_C(i) = V$ when $i = n$.
- For every $i \rightarrow_{\mathrm{CFG}} j$, $R^0_C(i)$ contains all variables v such that either (i) $v \in R^0_C(j)$ and $v \notin \mathrm{DEF}(i)$, or (ii) $v \in \mathrm{REF}(i)$, and $\mathrm{DEF}(i) \cap R^0_C(j) \neq \emptyset$.

$$S^0_C \equiv \{i \mid \mathrm{DEF}(i) \cap R^0_C(j) \neq \emptyset, i \rightarrow_{\mathrm{CFG}} j\}$$

$$B^k_C \equiv \{b \mid i \in S^k_C,\ i \in \mathrm{INFL}(b)\}$$

$$R^{k+1}_C(i) \equiv R^k_C(i) \cup \bigcup_{b \in B^k_C} R^0_{(b,\mathrm{REF}(b))}(i)$$

$$S^{k+1}_C \equiv B^k_C \cup \{i \mid \mathrm{DEF}(i) \cap R^{k+1}_C(j) \neq \emptyset, i \rightarrow_{\mathrm{CFG}} j\}$$

Taken from [Tip 95], see further reading

Interprocedural slicing

- Slicing across procedure boundaries
- First calculate slice in procedure containing C
- For procedure calls to Q use:
  - variables that may be modified by Q as Def(call Q)
  - variables that may be used by Q as Ref(call Q)
- Then calculate slices for all procedures that are called or call the original procedure
- Criterion:
  - Callee: (Last statement in called proc, relevant vars in P, in scope of called proc)
  - Caller: (Any call to P, relevant vars in first statement of P, in scope of calling proc)

Alternative Implementations

- Other implementations based on
  - Information flow relations
  - Dependence graphs
- Need to extend algorithms to cope with
  - Unstructured control flow (break, goto, etc)
  - Arrays, pointers and datatypes
  - Distribution and concurrency
- Algorithms vary in accuracy and efficiency, especially when dealing with above factors
- Also algorithms for dynamic and quasi-static slicing
- Language specific issues

Note on the Halting Problem

- Program Slicing in the most general case is undecidable
- Therefore define a slice as equivalent to the original program *only when the program terminates*
- Weiser also argues that calculating a *minimal* slice is undecidable
  - We can not find equivalence of two code fragments
  - But slices are small enough

Tools

- Mostly do simple, static slicing
- Advanced program slicing only so far implemented on toy languages
- Most are not comprehensive
- BUT still powerful and very useful

Tools

- Wisconsin Program-Slicing Tool/CodeSurfer
  - Multi-platform, C and C++
  - Forward and backward slicing, chopping. Static
- Unravel
  - C, only static backward slicing
- Bandera/Indus/Kaveri
  - Implements slicing as part of a tool set for model checking
  - Concurrent Java.
  - Eclipse plugin – multi-platform

Summary

Program Slicing:

- Reduces complexity for debugging and comprehension
- Filters statements that do not affect the values at a point of interest
- Many implementations
  - eg: dataflow analysis
- Tool support

Further Reading

- An Overview of Program Slicing. M Harman & R Hierons. Software Focus, 2001. *http://www.dcs.kcl.ac.uk/staff/mark/sf.html*
- Program Slicing. Mark Weiser. IEEE Transactions on Software Engineering, 1984.
- A Survey of Program Slicing Techniques. Frank Tip. Journal of Programming Languages, 1995.
- Program Slicing Literature Survey. Jeff Russel. *http://www.ece.utexas.edu/~jrussell/seminar/slicing_survey.pdf*. 2001
- Google!